# A HIGH-PERFORMANCE MULTITHREADED APPROACH FOR CLUSTERING A STREAM OF DOCUMENTS

**Janez Brank, Gregor Leban, Marko Grobelnik**
Artificial Intelligence Laboratory
Jožef Stefan Institute
Jamova 39, 1000 Ljubljana, Slovenia
Tel: +386 1 4773778; fax: +386 1 4251038
e-mail: {janez.brank,gregor.leban,marko.grobelnik}@ijs.si

**ABSTRACT**
*We present an efficient approach for clustering a massive stream of textual documents, with particular emphasis on parallelization by the means of multithreaded processing. The underlying clustering algorithm is adaptable to changes in the stream and includes the ability to split and merge clusters, as well as to discard old data.*

## 1 INTRODUCTION

Streams of textual documents occur naturally in various domains, such as from news sources (e.g. Spinn3r, IJS Newsfeed [4]) and social media (e.g. blogs, twitter, etc.). Clustering is the task of arranging such documents into groups based on some perceived notion of similarity, and is often an important building block of applications. The output of the clustering process can be seen as representing a "higher-level" view of the underlying data stream and is then the basis for further processing.

The fact that data comes from a stream adds a few specific requirements to the clustering task. In particular, the clustering algorithm must allow the partition of documents into clusters to be built and updated incrementally as new documents appear in the stream; it cannot make multiple passes through the entire stream, as there is only enough storage space for documents from a relatively limited time-based window. Additionally, we assume that the contents of the stream may change over time and the set of clusters should gradually be able to adapt to that, allowing the introduction of new clusters and the splitting/merging of existing clusters.

The most important consideration for a stream-based clustering algorithm, however, is that it must on average be able to process at least as many documents per unit of time as there are new documents coming from the stream in the same unit of time, otherwise it won't be able to keep up with the incoming data. If at some point the volume of incoming data grows high enough, parallelism will need to be employed. In the present paper, we focus on the scenario where the rate of incoming data is high enough that parallelism is necessary, but not so high that it would be necessary to distribute the processing over multiple computers. Thus, our aim is to present an efficient multi-threaded clustering approach that runs on a single computer but makes good use of its multi-CPU and multi-core facilities.

The remainder of this paper is structured as follows. In Section 2, we present an architectural overview of our clustering service. In Section 3, we briefly describe our clustering approach as it would appear conceptually if its multi-threaded aspects were taken out of consideration; in Section 4 we describe the design of the multi-threaded version of the approach; Section 5 presents conclusions and discusses possible directions of future work.

## 2 THE ARCHITECTURE OF OUR CLUSTERING SERVICE

Our implementation runs as a web service, listening for HTTP requests which provide new documents that arrived from the source stream and now need to be added to our clustering-related data structures. The clustering service processes such requests asynchronously; it reports any changes in the assignment of documents to clusters by eventually making HTTP requests to one or more listeners, reporting such things as assignments of documents to clusters, outcomes of cluster splits/merges, and changes in cluster medoids.

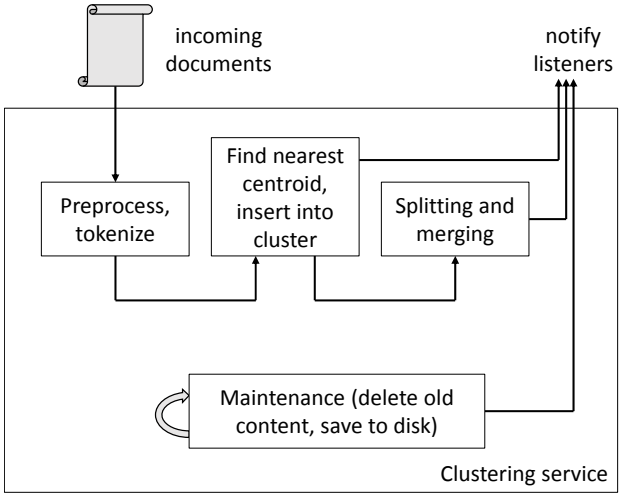


**Figure 1:** An architectural overview of our clustering web service.

In addition to handling the requests for adding a new document into the clustering, the service also performs maintenance operations from time to time. This includes discarding old documents and clusters, as well as saving its data structures to disk. The service keeps all its data in main memory during normal operation, but saves it to disk periodically so that it can be restarted without much loss of data in case it crashes. In earlier versions of our system, cluster merging was also performed periodically as a maintenance operation, but now it's included as a post-processing step following each addition of a new document.

## 3 THE UNDERLYING CLUSTERING ALGORITHM

Our approach is largely based on that of Aggarwal and Wu

[1][2]. The main principle that we use in assigning documents to clusters is to simply assign each incoming document to the cluster whose centroid is the closest (i.e. most similar) to the document. (Note that the same idea is used by popular off-line clustering algorithms such as k-means, except that k-means then performs multiple passes through the data, recalculating centroids and reassigning documents in each pass, which we cannot afford to do in an on-line setting.) For the purposes of computing centroids, measuring similarity etc., we use the bag-of-words representation (also known as the vector space model) to represent each document with a TF-IDF feature vector, normalized to unit length. After a new document is assigned to its nearest cluster, we consider splitting that cluster or merging it with another cluster. Figure 2 presents the pseudocode of our approach.

```
Input: d – a document to be added to the clustering
1   If d appears to be a duplicate of a document that is
    already in the clustering, stop processing d and ignore it.
2   Compute x, the TF-IDF vector representing d,
    normalized to unit length.
3   Find the cluster C whose centroid is the closest to x
    (in terms of cosine similarity).
4   Add d into C, updating its various aggregated statistics
    (such as the centroid).
5   If the splitting conditions are met, consider splitting C:
6       Find the most promising split of C into C' and C''.
7       If this split is better than the original C, replace C
        with C' and C'' in our clustering data structures.
8   Else, if the merging conditions are met, consider merging
    C with another cluster:
9       Find a few clusters whose centroid is closest to the
        centroid of C, in terms of cosine distance.
        For each such cluster C':
10          Let C'' = C ∪ C'. If this cluster is better than
            keeping C and C' then
11              Replace C and C' by C'' in our clustering data
                structures and break.
```

**Figure 2:** Pseudocode describing an overview of our clustering approach.

Following the approach of [1], we maintain a set of statistics for each cluster and update them incrementally whenever the cluster changes. This includes the sum of the feature vectors of its documents, the square of this sum, per-feature variances, and a few other statistics. Our implementation also supports the option of allowing different documents to have different weights, where the weight of the document decays exponentially as the document ages, as suggested by [1]; however, this has not been found to be useful in our applications, so we currently set all weights to 1 in practice.

The fact that we're dealing with an ever-changing stream of documents requires us to introduce a few approximations. For example, in principle, whenever a new document is added to the clustering, or an old document discarded, the document frequency (DF) of any term from that document changes; as a result, the inverse document frequencies (IDF) of such terms also change, and the value of their corresponding features change accordingly, in the TF-IDF vector of any document containing any such term, as well as in the centroid of any cluster containing any such document. So theoretically, the feature vectors of most documents and the centroids of most clusters would have to be recalculated whenever a document is added to or removed from our collection. Since the costs of such an update would be prohibitive, we introduce an approximation. First, instead of storing TF-IDF vectors of individual documents, we store TF vectors instead. The IDF can be applied on the fly whenever needed, e.g. when we wish to (re)calculate the centroid of a cluster. Secondly, when a document is added to a cluster, we only recalculate the centroid of that cluster, but not the centroids of other clusters; those will get recalculated sooner or later when some new document is added to them.

To save time, a cluster is only considered for splitting (line 5 of the algorithm listing) if it is sufficiently large and if sufficiently many additions to it have been made since the last time it has been considered for splitting. The main idea during splitting is to project all members of the cluster onto a line and divide them into two groups depending on whether their projection was left or right of the projection of the centroid. This is repeated several times; in the first iteration, we project onto the principal component of the original cluster; in each subsequent iteration, we project onto the line through the centroids of the two groups from the previous iteration. In line 6, the best of these splits is chosen based on minimizing the variance; in line 7, a Bayesian Information Criterion is used to decide whether to actually accept the split.

For merging, we similarly only consider the cluster for merging (in line 8) if enough additions have been made to it since the last time it was considered for merging. Merging makes sense if two clusters are similar, e.g. as measured by the cosine similarity of their centroids. The problem here is that while the feature vectors of individual documents are sparse (i.e. they have relatively few nonzero components), the centroid of a cluster is usually fairly dense. Thus, computing a cosine between two centroids involves a dot product of two dense vectors, which is time-consuming. We resort to an approximation again: for the purposes of step 9, we temporarily make the centroid of $C$ sparse by setting all its components to 0, except the thousand components that were highest in terms of absolute value. This substantially preserves the direction of the vector but makes the computation of dot products cheaper. In line 10, we use Lughofer's ellipsoid-overlap criterion [3] to decide whether to accept the merge; additionally, the merge is always accepted if the cosine between the two centroids exceeds a user-defined threshold.

The duplicate-detection step in line 1 is in practice somewhat custom-tailored to the particular document stream we've been using in our applications so far. This stream collects news articles from numerous websites, many of which turn out to be reprints of agency articles with few or no modifications. We declare an article to be a duplicate if an existing article has the same title (modulo capitalization and whitespace) and a sufficiently similar TF-vector. Such duplicate articles are simply discarded, rather than added into any cluster.

## 4 MULTI-THREADED CLUSTERING

A single-threaded, non-parallel implementation of the approach described in Section 3 is fairly straightforward. The program simply processes requests (to add a new document into the clustering) sequentially in an endless loop, finishing one request before moving on to the next one. Occasionally it can perform maintenance tasks (such as saving to disk, and discarding old clusters and documents) in between handling two requests.

Following the well-known principle that optimization should focus on those parts of the program in which the largest amount of time is spent, we timed the single-threaded implementation while performing $10^6$ article additions. It turns out that approx. 54% of the time was spent in step 3, calculating the cosine similarity between the new document and the centroids of all existing clusters; 43% of the time was spent in step 9, calculating the cosine similarity between cluster centroids; all other steps together account for the remaining 3% of the time. Thus, it is clear that parallelization needs to focus on steps 3 and 9.

Another important consideration when designing a multithreaded solution involves the use of shared data structures. If a thread needs to modify some shared data structure, it requires exclusive access to it; that is, other threads shouldn't be using the data structure at all while it's being modified, even if they are content with read-only access to it. At the same time, we want to minimize the amount of time that threads spend waiting for some other thread to relinquish its exclusive lock on a shared data structure.

In the algorithm from Figure 2, modifications of shared data structures occur in step 4 (adding a new document to a cluster), step 7 (performing a split) and step 11 (performing a merge). Another modification of shared data, which is not as readily obvious from that pseudocode listing, occurs when creating a feature vector **x** corresponding to the new document *d*: a shared hash table containing the document frequencies of all terms needs to be updated, and new words might need to be added into it (if *d* contains some words that have until now never been seen in our stream of documents). Similarly, the duplicate detection in step 1 uses a hash table of document titles and needs to add the new document's title to it (if it didn't turn out to be a duplicate).

A somewhat naïve idea would be to assign each newly incoming document to one of several threads, and this thread can then execute the algorithm from Figure 2. The thread could somehow lock the cluster while it's being accessed, but we can quickly see that this is unsatisfactory. Our application calls for a large number of small clusters; eventually there will be thousands, possibly tens of thousands of clusters, and we don't want to require a thread to have to acquire and release tens of thousands of locks during step 3 or step 9.

This sort of fine-grained locking also has other inconvenient aspects. It is easy to imagine a nightmare scenario in which one thread is trying to compute the cosine between the centroid of cluster *C* and a new document; another thread has already decided that it wants to insert a different new document into *C* and now wants to update its centroid; and yet another thread is trying to split *C* into two clusters, or merge it with some other cluster.

If we don't want to have to deal with cluster-level locking, we have to accept that no thread may modify clusters (which includes adding or deleting them) while some other thread is looping through them in step 3 (or step 9, for that matter). Thus, while any thread is modifying the shared data structure (i.e. performing steps 4, 7, or 11), no other thread may be reading this data (i.e. performing step 3 or 9 – but as we saw earlier, that's where our threads will be spending 97% of their time). For nearly every new document (unless it was discarded in step 1 as a duplicate), we'll eventually have to add it to a cluster (in step 4); before our thread can do so, it must wait for all other threads to reach the end of step 3 or 9, and all those threads must then stop and wait for our thread to finish step 4. (A similar consideration applies to steps 7 and 11, but those are performed more rarely.) Clearly this has the potential to lead to an undesirable amount of waiting.

We can rewrite the pseudocode of Figure 2 in a way which emphasizes the alternation between stages which only read shared data and stages which need to modify shared data:

R1. Check if *d* is a duplicate; split it into words and prepare a TF-vector, except for any new words that aren't in our shared word table yet – these should be kept in a separate list.

M1. If R1 found *d* to be a duplicate, discard it and stop. Otherwise, add its title to the shared hash table (for future duplicate detection) and add any new words it might have contained into the shared word table; this is also the time to finalize its TF vector and update the document frequency counts in the shared word table.

R2. Compute the TF-IDF vector of our document *d* and find the cluster *C* with the nearest centroid.

M2. Insert *d* into *C*, updating *C*'s centroid and other aggregate statistics.

R3. Consider splitting *C* or merging it with other clusters, if appropriate. Do not modify any shared data structures; if the decision to split *C* is made, record what the new clusters *C'* and *C''* would be; likewise, if the decision for a merge is made, record which cluster it would merge with and what would the resulting cluster be like.

M3. Update the shared data structures to reflect the splits or merges decided upon in step R3.

The key observation here is that there is no reason why all these steps should be performed by the same thread, as long as we maintain a small "context" data structure which helps threads keep track of the request as it passes through the stages.

Note also that stage R2 basically corresponds to step 3 of Figure 1, while step 9 is included within R3. All the M-stages are cheap, quick operations. Thus, in our multithreaded clustering implementation, we have *a single main thread which performs the M-stages for all requests*; the other threads are worker threads and perform R-stages.

The requests pass between the main thread and the worker threads until they are complete, as shown on Figure 3.

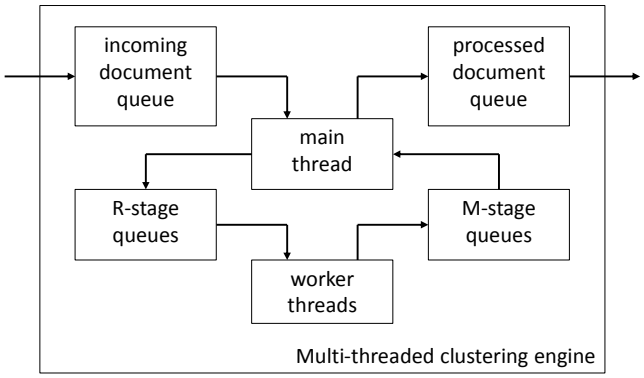


**Figure 3:** An overview of our multi-threaded clustering approach, showing the flow of requests through the system.

Thus, each worker thread runs in an endless loop in which it takes a job from a queue, performs the next stage (which will be either R1, R2, or R3), and deposits it into a different queue. The main thread, on the other hand, assigns jobs and periodically blocks the worker threads, performs the M-stages, and restarts them. The following is a simplified pseudocode of the main thread:

1 Wait a set amount of time (e.g. 1 second).
2 Set a flag which tells the worker threads to stop after they finish their current job. Wait for all the worker threads to finish their current job.
3 Perform the M-stages of all the requests which are currently in progress.
4 Return to step 1.

Thus, most of the time the main thread sleeps (step 1) and lets the worker threads perform the R-stages of various requests. Every now and then, the main thread performs a barrier synchronization (step 2), stopping all the workers after their current job is done. Thus, at step 3, all workers are asleep, so the main thread can modify shared data.

Occasionally (e.g. once per hour), the main thread stops issuing any new R1-stage jobs to worker threads and waits for all partly completed requests to fully complete (i.e. all the way through M3). At this point, there are no partly completed requests in the system, so this is a good time to perform periodic maintenance tasks such as saving the data to disk. After this, normal processing can resume.

Since step 3 performs the M-stages of all currently open requests in one place, it is in a good position to coordinate their sometimes conflicting ideas as to what should be done. First, it performs the M1-stages, as these cannot conflict with other requests. Next it performs any splits and merges (M3) requested by recently completed R3 stages; while doing so, the main thread keeps track of which clusters have been split or merged, and ignores split/merge requests that involve clusters that have been affected by a previously processed split/merge request. Finally, the main thread performs M2-stages, inserting documents into the clusters requested by recently completed R2 stages; if any such cluster has been split, the main thread checks the centroids of the two subclusters to see which is closer to the document.

Note that this approach means that each document must pass through three iterations of the main thread before it is fully processed. Thus, if e.g. step 1 of the main thread takes 1 second, it will take at least 3 seconds before the document is processed. To use an analogy from networking, we have achieved high bandwidth at the price of also having high latency.

## 5 CONCLUSIONS AND FUTURE WORK

The multithreaded clustering approach presented here achieves a considerable degree of parallelism, allowing it to fully utilize a typical present-day multi-CPU multi-core PC. A further form of parallelism, not mentioned above but present in our application, comes from the fact that our stream of documents is multilingual and each language is processed separately from the others, thus each language can have its own main thread and set of worker threads.

The work presented here could be extended in several directions. For example, this approach could be applied to non-textual data with only minor modifications. The key idea behind our multithreaded approach is the multi-stage processing, concentrating all modification of shared data into one thread and using barrier synchronization for the worker threads; and there is nothing text-specific in this.

The computation of cosine similarities (which is where the algorithm still spends most of its time) could be speeded up by the means of random projections [5] into a limited (and fixed) number of dimensions. In our preliminary experiments, projecting our feature space into 1000 random projections resulted in almost no distortion (in terms of which centroid is closest to which document). After such a projection, documents and centroids become fixed-length dense vectors, and cosines can be computed very efficiently by making use of the SIMD capabilities of modern processors (as in various numerical linear algebra libraries).

Another possible direction for further work is to replace the current flat clustering with a hierarchical one. This can be desirable for some applications, and it would also speed up the assignment of documents to clusters if this is done in a top-down fashion instead of examining all the clusters.

Finally, at some point the rate of incoming documents may well grow beyond what can be processed by a single computer, so it would be interesting to investigate clustering approaches based on distributed computing.

**Acknowledgments**

This work was supported by the Slovenian Research Agency and the ICT Programme of the EC under under XLike (ICT-STREP-288342).

**References**

[1] C. C. Aggarwal, P. S. Yu. A framework for clustering massive text and categorical data streams. *SIAM Conf. on Data Mining*, 2006.

[2] C. C. Aggarwal, P. S. Yu. On clustering massive text and categorical data streams. *Knowledge and Inf. Systems*, 24(2):171–196, 2010.

[3] E. Lughofer. A dynamic split-and-merge approach for evolving cluster models. *Evolving Systems*, 3(3):135–151, 2012.

[4] M. Trampuš, B. Novak. Internals of an aggregated web news feed. *Proc. SiKDD 2012*.

[5] Â. Cardoso, A. Wichert. Iterative random projections for high-dimensional data clustering. *Pattern Recognition Letters*, 33(12):1749–55, 2012.